AMD's Direct3D Optimizations

Kent Knox
Software Research & Development
DirectX Team

AMD's DirectX History

- DirectX6
  - Fixed function pipeline – Optimized for the K6 family processors with 3dNow! technology
- DirectX7
  - Fixed function pipeline – Optimized for the Athlon and the K6 family processors with 3dNow! technology
- DirectX8
  - Fixed function pipeline – Optimized for the Athlon and the K6 family processors with 3dNow! technology
  - Vertex shader pipeline – Optimized for Athlon XP and older processors with SSE and 3dNow! technology
  - D3DX8 – Optimized for Athlon and K6 family processors with 3dNow! technology

AMD's DirectX History cont.

- DirectX9
  - 32-bit
    - Fixed function pipeline – Optimized for the Athlon and the K6-2 processors with 3dNow! technology
    - Vertex shader pipeline – Optimized for Athlon XP and older processors with SSE and 3dNow! technology
    - D3DX9 – Optimized for Athlon and K6 family processors with 3dNow! Technology; new `*array` routines especially efficient for SIMD processing
  - 64-bit
    - Fixed function pipeline – Optimized through Vertex shader emulation layer; fixed function really is a big shader
    - Vertex shader pipeline – Optimized for Athlon FX and older processors with SSE and SSE2 technology; no 3dNow! technology available
    - D3DX9 – No 64-bit optimizations yet. Takes native advantage of extra registers and improved calling stack

How to enable Software Vertex Shaders

- Microsoft API D3D Create Device call accepts flags to determine what kind of vertex processing will be done; passing the `D3DCREATE_SOFTWARE_VERTEXPROCESSING` flag to the `CreateDevice` call will create the Software VM.
- It is possible to mix hardware and software vertex processing within the same app; the `CreateDevice` call accepts the `D3DCREATE_MIXED_VERTEXPROCESSING` flag. It is important to keep track of where buffers are created for optimum performance.
- It's important to remember that for software vertex processing (SVP), the application should create all buffers (Index/Vertex) in system memory as opposed to video memory.

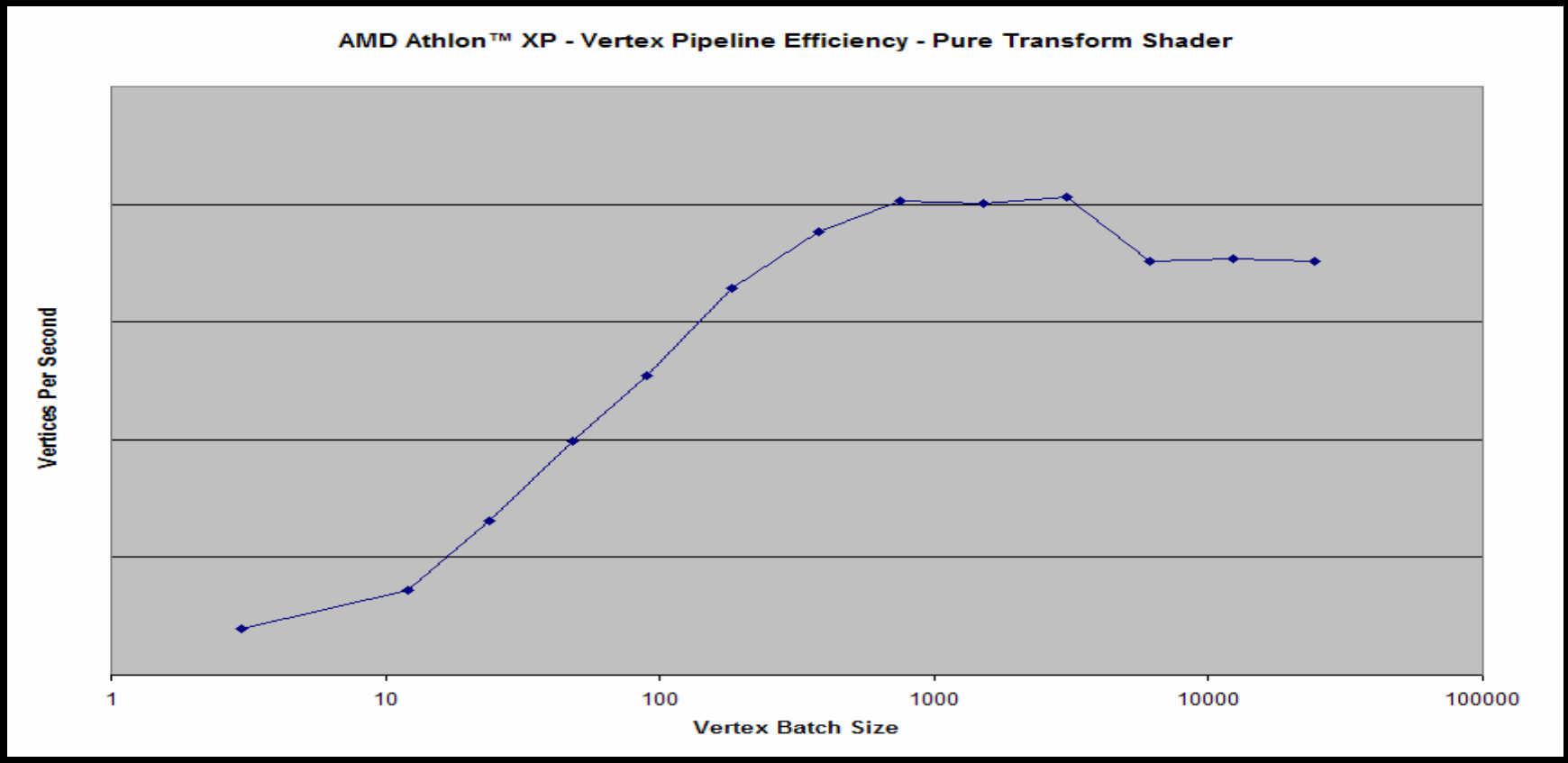
Software Vertex Processing Optimizations

AMD

- The optimizations listed below only apply to vertex shaders. Pixel shaders are always computed in hardware and are excluded from software optimizations.
- These optimizations apply mostly to when an application developer writes their shaders by hand. Use of a high level shader language does not typically allow this level of control. However, these optimizations could be picked up by the compilers themselves.
- Most of these optimizations apply to both hardware and software vertex processing. There are a few non-obvious cases when something should be done differently in software than hardware, and are noted below.

Software Optimization #1

- Batch up vertices and send as large a batch down the graphics pipeline as possible in one `Draw*Primitive` call.

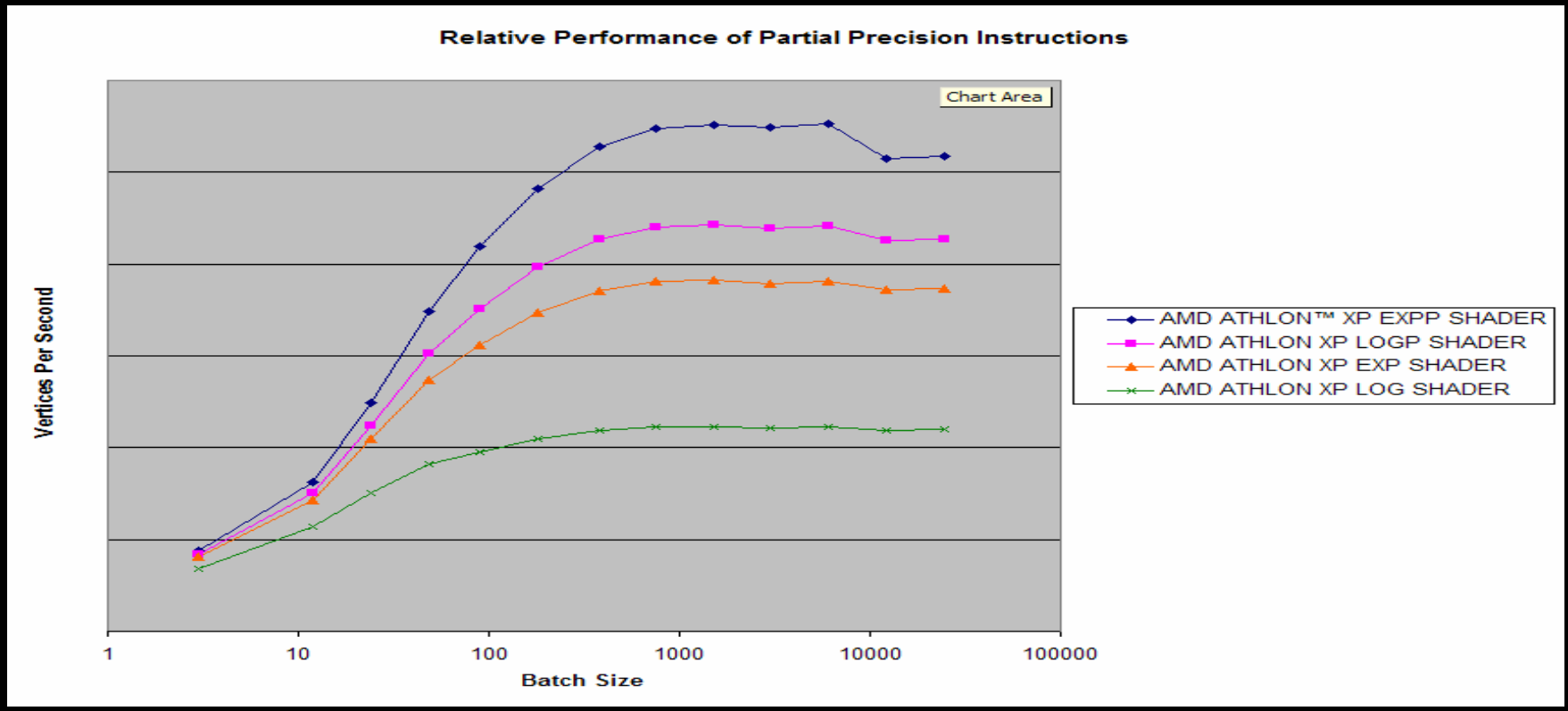


Software Optimization #2

- DirectX9 contains the first implementation of AMD's code that is multi-processor aware.
  - AMD's code detects the presence of a second (or Nth) processor, and enables support automatically.
  - The second processor is only utilized if the size of the batch (in vertices) sent down the pipeline exceeds a pre-defined threshold. Otherwise, the overhead of splitting the batch into N-batches and synchronization between the threads will exceed the speedup. Ex: AMD's code will not split a batch of 3 vertices between 2 processors.
  - AMD has determined experimentally that a batch needs to contain near 700 vertices before a second processor becomes a benefit. This is a another good example of why it is beneficial to batch up vertices (software optimization #1).

Software Optimization #3

- Partial precision instructions are faster than full precision instructions. Use these instructions to calculate lighting; typically lighting does not need to be fully precise, as color values are eventually quantized into 1 of 256 values.



Software Optimizations #4 & #5

- Macro instructions such as `m4x4` and `m3x3` allow the software virtual machine to detect that a matrix multiplication is present, and to generate optimized code for those routines. Use macro instructions when possible, including `NRM`, `CRS`, …
- Writemasks are important to vertex shaders to eliminate not-needed work. If only two components of a register are used in the program, use a writemask on the instruction to specify only those two components.
  - In practice, AMD's software virtual machine analyses and optimizes all shaders submitted to it, and eliminates any useless work (dead code elimination). However, it is a good habit to be in, and the smarter the shader is the less it depends on the "smarts" of any driver code beneath it.

Software Optimization #6

- DirectX9 has introduced many new data types for vertices. Avoid using the new float16 data types when it is known that the software virtual machine will be used.
  - The smallest float type that processor hardware supports is a 32-bit float. AMD's virtual machine must expand out 16-bit floats to 32-bit floats before processing can begin, and the conversion is non-trivial. As a result from the necessary conversion, any benefit from the smaller float size is negated.
  - This is a significant deviation from optimizing for graphics hardware cards, which benefit from 16-bit floats because of reduced register pressure and smaller AGP transfers.
  - D3DX9 provides optimized routines to convert from 32 to 16-bit floats and vice versa. If the app's source data is a 32 bit stream:
    - For software processing; pass the data as is
    - For hardware processing; convert data (i.e. texture data) to 16-bit before passing down

Software Optimization #7 & #8

- For vs.3.0 shaders, the `texldl` instruction is not optimized in AMD's virtual machine.
  - This instruction is implemented via a software callback to un-optimized reference C code.
  - Expect performance in software to drop with any shader that uses this instruction.
- The `CreateShader` call is when we JIT compile the shader into SIMD processor code. Avoid creating shaders during the play of the game; instead create all shaders during level load or some other convenient non real-time portion of the game. Cache the shaders and use them when you need them.

Software Optimization #9

- Dynamic conditional branches (`ifc`, `ifp`, `breakc`, `breakp`) are tricky to use efficiently. They can sometimes hurt, even if they are used to conditionally skip code. The trick is to ensure that the speedup gained from skipping execution of the conditional code is greater than the fixed overhead cost imposed by the conditional branch.
  - This is due to the nature of the software VM, which uses the SIMD architecture of the processor (either 3dNow! or SSE) to calculate vertex transformations. Up to 4 vertices could be in flight at the same time, and a conditional branch may cause different vertices to go through different codepaths. Extra housekeeping code needs to be generated to make sure that all vertices are in sync with each other after the branch.
  - Some rules of thumb:
    - If the body of the conditional code is no more than a few simple (non-macro) instructions, it is better to use predication to conditionally execute code. Predication helps to solve this sync problem because it is atomic to the instruction.
    - If the body of the conditional code contains more than 4 instructions and/or contains macro instructions, leave the conditional code as is. A good example of this is an early out for a cone light. The benefit of skipping the `lit` instruction far outweighs the overhead of the branch.

Software Optimization #9 Cont.

- Example of rewriting code to use predicates:
- Before:

```
if_gt  r0.x, c[9].x              ; skip if negative
  mul r1.xyz, r0.x, c[aL+5]      ; scale diffuse color
  add r1.xyz, r1, c[aL+5]        ; add ambient
  mad r10.xyz, r1, r2.x, r10     ; attenuate color and add
endif                            ; if_gt  r0.x, c[9].x
```

- After:

```
setp_lt p0.x, r0.x, c[9].x
(p0.x) mul r1.xyz, r0.x, c[aL+5]
(p0.x) add r1.xyz, r1, c[aL+5]
(p0.x) mad r10.xyz, r1, r2.x, r10
```

Software Optimization #9 Cont.

- Example of code to leave inside a conditional branch:

```
if_gt  r0.x, c[9].x                ; skip if negative
  add r3.xyz, c[8], -v0            ; vector from vertex to eye
  nrm r4, r3                       ; normalize
  add r1.xyz, r1, r4               ; r1 = half vector
  nrm r4, r1                       ; normalize
  dp3 r0.y, v1, r4                 ; dot normal with half vector
  mov r0.w, c[7].x                 ; get specular power
  lit r3, r0                       ; calculate lighting
  mul r1.xyz, r3.y, c[aL+5]        ; scale light diffuse color
  add r1.xyz, r1, c[aL+5]          ; add ambient
  mad r10.xyz, r1, r2.x, r10       ; attenuate diffuse color
  mul r1.xyz, r3.z, c[aL+5]        ; scale light specular color
  mad r11.xyz, r1, r2.x, r11       ; specular colors
endif                              ; if_gt  r0.x, c[9].x
```

Software Optimization #10

- Minimize use of the a0 register whenever possible. The a0 register is an indirect reference to the register file that the VM can not resolve at compile time. This register significantly complicates access to the register file, and can measurable slow access time down.

- Un-optimized code:

```
dp4 r0.x, v0, c[a0.x+1]      ; transform position vector
dp4 r0.y, v0, c[a0.x+2]
dp4 r0.z, v0, c[a0.x+3]
dp4 r1.x, v1, c[a0.x+1]      ; transform normal vector
dp4 r1.y, v1, c[a0.x+2]
dp4 r1.z, v1, c[a0.x+3]
```

Software Optimization #10 Cont.

- Optimized code:

```
mov r2, c[a0.x+1]        ; pre-load contents of a0 indirection
mov r3, c[a0.x+2]
mov r4, c[a0.x+3]
dp4 r0.x, v0, r2         ; transform position vector
dp4 r0.y, v0, r3
dp4 r0.z, v0, r4
dp4 r1.x, v1, r2         ; transform normal vector
dp4 r1.y, v1, r3
dp4 r1.z, v1, r4
```

Suggestions for Software shader use

AMD

- Prototyping; AMD will support all vertex shader models on introduction of a new DirectX. AMD has had an optimized version of vs_3_0 shaders since 12/2002; hardware is just now catching up.
- vs_(2/3)_sw shaders; a number of restrictions inherent to vertex shaders are relaxed in vs_(2/3)_sw shaders; this could be useful for debugging purposes.
  - Unlimited # of instructions
  - Float constants are increased to 8192
  - Integer and Boolean constants increased to 2048
  - Unlimited number of flow control instructions; up to 2048 labels are supported
  - Output registers file increased to 16
  - No port limits on any of the register files
  - Loop initial value and iteration step/stride increased to 32-bit numbers

Common Misconceptions about Shaders

AMD

- "My game needs to be able to support Dx7 class hardware. We looked at shaders and think they're cool, but we have requirements to support fixed function".
  - With software vertex shaders, hardware does NOT have to support shaders (Dx7 class hardware). The game can be written to support the latest video cards with hardware shaders, and can fall back gracefully to software shaders for Dx7 hardware.
- "My game needs to be able to access the post-T&L vertices to be able to implement feature 'AlphaBetaGammaZulu' in my game. We fall back to fixed function to do this".
  - No need to do this. Microsoft provides the `ProcessVertices` API to transform vertices from one buffer to another without drawing to the card. `ProcessVertices` is guaranteed to execute in software, no matter what current mode the Direct3D device is in.

D3DX Optimizations

AMD

- It really is very fast! Use D3DX instead of rolling your own math helper routines.
- All D3DX optimizations are implemented using the 3dNow! instruction set, except for the `*array` routines introduced in D3DX9 which were implemented with the SSE instruction set.
- For really fast processing of batches of vectors, D3DX9 introduced the `*array` routines. These routines are designed to take advantage of the SIMD nature of SSE.
  - On a batch of 10 vectors, it is faster to call `D3DXVec4TransformArray` once than it is to call `D3DXVec4Transform` 10 times.
  - If less than 4 (natural SSE width) vectors needs to be transformed, it is faster to call the non-array version.

Questions?

Web Links:
www.microsoft.com/directx
developer.nvidia.com
www.ati.com/developer

Special thanks to my team:
Jim Conyngham
Mark Santaniello
Navreet Gill

Contact Information:
kent.knox@amd.com
Austin, TX

Trademark Attribution